Second

Multics Programmers' Manual
Revision 12
November 30, 1972

Notes on this Update

With this update, a major revision of the organization and updating of the Multics Programmers' Manual occurs. The two parts of the manual, previously issued in a single volume, are with this revision divided into two volumes, each with separate title page and table of contents. Both volumes are currently considered to be at update level 12, but future updates will apply to one volume at a time.

In addition, the previously limited distribution "Subsystem Writers' Supplement" to the MPM is being reissued as volume III of the MPM, for general distribution. This third volume contains those subroutines and interfaces which are usually of interest only to the compiler writer or to the constructor of a protected subsystem. Since not all users will require volume III, it is not included in this update, but will instead be available for purchase at the I.P.C. publications office, approximately four weeks after the availability of this update.

Six very obsolete reference guide sections are deleted from Part II by this update, with no replacement in Part II; these sections (on fault assignment, linkage and stack formats, binding, and calling sequences) will appear in up-to-date form in the new volume III.

Probably the most significant new item appearing in this update is a new chapter 4, an introduction to programming in the Multics environment, with examples. This chapter fills a gap in Multics documentation which has been a particular problem for beginners.

In addition, three new and updated reference guide sections are included, leaving only two sections (under Procedures Which Should Not Be Called, following the subroutine write-ups) with the annotation "(old)".

(END)

1

MULTICS PROGRAMMERS' MANUAL

Revision 12

Changes from 9/1/72 to 11/30/72

Filing	Instructions
replace	Title Page
replace	Foreword (after Preface)
replace	Contents (after Foreword)
delete	II.3.1 (Naming Conventions) after I.2.10
delete	II.3.4.4 (Access Control) after I.2.10
delete	II.4.4 (Basic Addressing Techniques) after I.2.10
add	Chapter 4 (Programming in the Multics Environment) after chapter 3

Now, if you wish, you may separate everything up to and including chapter 4, and file it in a separate book as MPM Part I: Introduction.

add	Title Page, Part II
add	Foreword, Part II
add	Contents, Part II (after Foreword)
add	1.5 (Constructing and Interpreting Names) after 1.4
delete	1.5.1 (Hardware Feature to Avoid) after 1.5
delete	1.5.2 (Fault Assignment) after 1.5
delete	1.5.3 (Simulated Faults) after 1.5
delete	2.3 (Standard Call) after 2.2
delete	2.4 (Short Call) after 2.2
add	2.5 (System Programming Standards) after 2.2
delete	3.4 (Linkage Section) after 3.3
add	3.4 (Access Control) after 3.3
replace	Index (after Privileged Procedures)

(END)

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

The Multiplexed Information and Computing Service: Programmers' Manual

PART I

INTRODUCTION TO MULTICS

Revision: 12

Date: 11/30/72

FOREWORD

QA76
.5
M83
1972
Math/
Stat.

PLAN OF THE MULTICS PROGRAMMERS' MANUAL

November 30, 1972

The Multics Programmers' Manual (MPM) is the primary reference manual for user and subsystem programming on the Multics system. It is divided into three major parts:

Part I: Introduction to Multics

Part II: Reference Guide to Multics

Part III: Subsystem Writers' Guide to Multics

Part I is an introduction to the properties, concepts, and usage of the Multics system. Its four chapters are designed for reading continuity rather than for reference or completeness. Chapter 1 provides a broad overview. Chapter 2 goes into the concepts underlying Multics. Chapter 3 is a tutorial guide to the mechanics of using the system, with illustrative examples of terminal sessions. Chapter 4 provides a series of examples of programming in the Multics environment.

Part II is a self-contained comprehensive reference guide to the use of the Multics system for most users. In contrast to Part I, the Reference Guide is intended to document every detail and to permit rapid location of desired information, rather than to facilitate cover-to-cover reading.

Part II is organized into ten sections, of which the first eight systematically document the overall mechanics, conventions, and usage of the system. The last two sections of the Reference Guide are alphabetically organized lists of standard Multics commands and subroutines, respectively, giving details of the calling sequence and the usage of each.

Several cross-reference facilities help locate information in the Reference Guide:

- The table of contents, at the front of the manual, provides the name of each section and subsection and an alphabetically ordered list of command and subroutine names.
- A comprehensive index (of Part II only) lists items by subject.
- Reference Guide sections 1.1 and 2.1 provide lists of commands and subroutines, respectively, by functional category.

Part III is a reference guide for subsystem writers. It is of interest to compiler writers and writers of sophisticated subsystems. It documents user-accessible modules which allow a user to bypass standard Multics facilities. The interfaces thus documented are a level deeper into the system than those required by the casual user.

Examples of specialized subsystems for which construction would require reference to Part III are:

1) a subsystem which precisely imitates the command environment of some system other than Multics (e.g., an imitation of the Dartmouth Time-Sharing System);

2) a subsystem which is intended to enforce restrictions on the services available to a set of users (e.g., an APL-only subsystem for use in an academic class);

3) a subsystem which is protecting some kind of information in a way not easily expressible with ordinary access control lists (e.g., a proprietary linear programming system, or an administrative data base system which permits access only to program-defined aggregated information such as averages and correlations).

Each of the three parts of the MPM has its own table of contents and is updated separately, by adding and replacing individual sections. Each section is separately dated, both on the section itself, and in the appropriate table of contents. The title page and table of contents are replaced as part of each update, so one can quickly determine if his manual is properly up-to-date. The Multics on-line "message of the day" or local installation bulletins should provide notice of availability of new updates. In addition, the Multics command "help mpm" provides on-line information about known errors and the latest MPM update level.

In addition to this manual, users who will write programs for Multics will need a manual giving specific details of the language they will use; such manuals are currently available for PL/I, FORTRAN, and BASIC. A separate, specialized supplement to the MPM is also provided for users of graphic displays. The bibliography at the end of Part I, Chapter 1, describes these and other references in more detail.

Multics provides the ability for a local installation to develop an installation-maintained or author-maintained library of commands and subroutines which are tailored to local needs. The installation may also document these facilities in the same format as used in the MPM; the user can then interfile these locally provided write-ups in the command and subroutine sections of his MPM.

Finally, access to Multics requires authorization. The prospective user must negotiate with the administration of his local installation for permission to use the system. The installation may find it useful to provide the new user with a documentation kit describing available documents, telephone numbers, operational schedules, consulting services, and other local conventions.

(END)

CONTENTS

November 30, 1972

Contents

(END)

CHAPTER 4

PROGRAMMING IN THE MULTICS ENVIRONMENT*

September 29, 1972

A programmer may, if he wishes, treat Multics as simply a PL/I, FORTRAN, APL, BASIC, or LISP machine, and contain his activities to just the features provided in his preferred programming language. On the other hand, much of the richness of the Multics programming environment involves use of system facilities for which there are no available constructs in the usual languages. To use these features, it is generally necessary to call upon library and supervisor subroutines. Unfortunately, a simple description of how to call a subroutine may give little clue to how it is intended to be used. The purpose of this chapter is to illustrate typical ways in which one utilizes many of the properties of the Multics programming environment.

The programmer choosing a language for his implementation should carefully consider the extent to which he will want to go beyond his language and use system facilities of Multics which are missing from his language. As a general rule, one may say that each of the Multics languages matches some well-known standard for completeness of that language (e.g., ANSI or IBM). However, in going beyond the standard languages, the programmer will find that Multics tends to be biased towards convenience of the PL/I programmer. For example, if one plans to write programs which directly call the Multics storage system privacy and protection entries, he will be asked to supply arguments which are, in PL/I, structures. If he is writing in FORTRAN or BASIC, he has no convenient way to express such structures. Note that the situation is not hopeless, however. Programs which stay within the original language can be written with no trouble. Also, in many cases, one can construct a trivial PL/I interface subroutine, callable from, say, a FORTRAN program and which goes on to reinterpret arguments and invoke the Multics facility

---

* Note: All examples in this chapter use the "Version II" Multics PL/I compiler. Most comments, except those relating to the "profile" feature, also apply to the version I compiler.

system uses the Multics File Manager (2, above) very large files can be efficiently set up, updated, and searched using only the PL/I language. For further information, one should consult the PL/I language specifications.

In addition, users with unusually sophisticated needs such as completely inverted files, files with indexes on different elements, etc., will find that appropriate facilities can easily be developed using the virtual memory combined with techniques similar to those used by the Multics File Manager. It is important to realize that the Multics File Manager, while organized as a protected subsystem, is written in PL/I, using only Multics facilities which are also available to the user. Thus, a user could construct his own version of the File Manager, or a more elaborate file accessing system without recourse to special privileges or need to modify the Multics supervisor.

Finally, the Multics I/O system, which is organized to allow attachment of arbitrary source-sink I/O devices, may be used to read and write magnetic tape in any of several formats, for applications in which permanent on-line storage is not appropriate.

Unfortunately, there does not yet exist a suitable set of annotated case studies on the use of the file management facilities. The potential developer of a large file application is advised to begin by reviewing one or more applications previously implemented on Multics and which use these tools.

(END)*

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

The Multiplexed Information and Computing Service: Programmers' Manual

PART II

REFERENCE GUIDE TO MULTICS

Revision: 12

Date: 11/30/72

FOREWORD

PLAN OF THE MULTICS PROGRAMMERS' MANUAL

November 30, 1972

The Multics Programmers' Manual (MPM) is the primary reference manual for user and subsystem programming on the Multics system. It is divided into three major parts:

Part I: Introduction to Multics

Part II: Reference Guide to Multics

Part III: Subsystem Writers' Guide to Multics

Part I is an introduction to the properties, concepts, and usage of the Multics system. Its four chapters are designed for reading continuity rather than for reference or completeness. Chapter 1 provides a broad overview. Chapter 2 goes into the concepts underlying Multics. Chapter 3 is a tutorial guide to the mechanics of using the system, with illustrative examples of terminal sessions. Chapter 4 provides a series of examples of programming in the Multics environment.

Part II is a self-contained comprehensive reference guide to the use of the Multics system for most users. In contrast to Part I, the Reference Guide is intended to document every detail and to permit rapid location of desired information, rather than to facilitate cover-to-cover reading.

Part II is organized into ten sections, of which the first eight systematically document the overall mechanics, conventions, and usage of the system. The last two sections of the Reference Guide are alphabetically organized lists of standard Multics commands and subroutines, respectively, giving details of the calling sequence and the usage of each.

Several cross-reference facilities help locate information in the Reference Guide:

- The table of contents, at the front of the manual, provides the name of each section and subsection and an alphabetically ordered list of command and subroutine names.
- A comprehensive index (of Part II only) lists items by subject.
- Reference Guide sections 1.1 and 2.1 provide lists of commands and subroutines, respectively, by functional category.

Part III is a reference guide for subsystem writers. It is of interest to compiler writers and writers of sophisticated subsystems. It documents user-accessible modules which allow a user to bypass standard Multics facilities. The interfaces thus documented are a level deeper into the system than those required by the casual user.

Examples of specialized subsystems for which construction would require reference to Part III are:

1) a subsystem which precisely imitates the command environment of some system other than Multics (e.g., an imitation of the Dartmouth Time-Sharing System);

2) a subsystem which is intended to enforce restrictions on the services available to a set of users (e.g., an APL-only subsystem for use in an academic class);

3) a subsystem which is protecting some kind of information in a way not easily expressible with ordinary access control lists (e.g., a proprietary linear programming system, or an administrative data base system which permits access only to program-defined aggregated information such as averages and correlations).

Each of the three parts of the MPM has its own table of contents and is updated separately, by adding and replacing individual sections. Each section is separately dated, both on the section itself, and in the appropriate table of contents. The title page and table of contents are replaced as part of each update, so one can quickly determine if his manual is properly up-to-date. The Multics on-line "message of the day" or local installation bulletins should provide notice of availability of new updates. In addition, the Multics command "help mpm" provides on-line information about known errors and the latest MPM update level.

In addition to this manual, users who will write programs for Multics will need a manual giving specific details of the language they will use; such manuals are currently available for PL/I, FORTRAN, and BASIC. A separate, specialized supplement to the MPM is also provided for users of graphic displays. The bibliography at the end of Part I, Chapter 1, describes these and other references in more detail.

Multics provides the ability for a local installation to develop an installation-maintained or author-maintained library of commands and subroutines which are tailored to local needs. The installation may also document these facilities in the same format as used in the MPM; the user can then interfile these locally provided write-ups in the command and subroutine sections of his MPM.

Finally, access to Multics requires authorization. The prospective user must negotiate with the administration of his local installation for permission to use the system. The installation may find it useful to provide the new user with a documentation kit describing available documents, telephone numbers, operational schedules, consulting services, and other local conventions.

(END)

CONTENTS

November 30, 1972

Section 5 Standard Data Formats and Codes

Section 6 Handling of Unusual Occurrences

Section 7 Special Subsystems

Section 8 Miscellaneous Reference Information

Section 9 Commands and Active Functions (03/20/72)

continued on next page

Section 9 Commands and Active Functions (continued)

continued on next page

Section 9 Commands and Active Functions (continued)

Section 10 Subroutines (03/24/72)

continued on next page

Section 10 Subroutines (continued)

continued on next page

Contents

Section 10 Subroutines (continued)

<u>Procedures Which Should Not Be Called</u>

(END)

CONSTRUCTING AND INTERPRETING NAMES

The various types of names used on Multics are constructed and interpreted according to certain conventions. The names in question are user names, segment names, command names, subroutine names, I/O stream names and condition names.

User names are discussed in the MPM Reference Guide section, Access Control, since they are primarily used to specify access control information.

A segment may be named in two ways. Its location in the storage system hierarchy is specified by its path name. The name by which it is known in a process is its reference name. The star convention and equals convention provide short hand methods of specifying segment names. Offset names allow specification of externally known locations in a segment.

Path Names

As described in the MPM Introduction Chapter 3, Beginner's Guide to The Use of Multics, each segment (or directory or link) in the Multics storage system has an entry in a superior directory. Any segment (or directory or link) may be found by following the appropriate entries from a designated directory through inferior directories until the desired segment (or directory or link) entry is reached. An absolute path name is just such a sequence of entry names starting from the root directory. A relative path name is a sequence relative to the current working directory. Path names, whether relative or absolute, are typically used as arguments to commands and subroutines.

An entry name is a string of 32 or fewer ASCII characters. Only the greater-than (>) and less-than (<) characters are not allowed in entry names, since they are used to form path names as described below. Several other characters are not recommended for entry names -- asterisk (*), equals (=) and dollar sign ($) -- because standard commands attach special meanings to them. Each is explained below.

In general, entry names will consist of the upper- and lower-case alphabetic characters, the digits, the underscore (_) and the period (.), and must have at least one nonblank character. The underscore is used to simulate a space for readability; e.g., a segment might be named new_seg. (Including a space in an entry name is permitted, but is cumbersome since the command language uses spaces to delimit command names and arguments.) The period is used to separate components of an

entry name, where a component is a logical part of the name. Several system conventions depend on components. For example, compilers on Multics expect the language name to be the last component of the name of a source segment to be compiled; e.g., square_root.pl1 for a PL/I source segment.

An absolute path name is formed from a sequence of entry names, each preceded by a greater-than character. The initial greater-than indicates that the entry name following it designates an entry in the root directory. Thus, an absolute path name has the form >first_dir>second_dir>third_dir>my_seg.

The directory first_dir is immediately inferior to the root, second_dir is an entry in first_dir, etc. A maximum of 16 levels of directories is allowed from the root to the final entry name. The number of characters in the path name may not exceed 168. Each intermediate entry in the chain may be either a directory or a link to a directory. The final entry may be a directory, a segment or a link.

A relative path name looks like an absolute path name except that it does not contain a leading greater-than character, and may begin with less-than characters as explained below. It is interpreted by various commands to be a path name relative to the user's working directory. The simplest form of relative path name is the single name of an entry in the user's working directory. For example, the relative path name alpha refers to the entry alpha in the user's working directory. On a slightly more complex level, the relative path name sub_dir>beta refers to the entry beta in the directory sub_dir which is immediately inferior to the user's working directory.

The less-than character may be used at the front (left end) of a relative path name to indicate that the directory immediately superior to the working directory is where the following entry name is to be found. This principle may be extended so that several less-than characters cause the superior directory several levels higher than the working directory to be searched for the first entry name in the relative path name.

In the following examples, the user's working directory is

>dir1>dir2>dir3>dir4

A relative path name of

new_seg

would designate the segment with the absolute path name

>dir1>dir2>dir3>dir4>new_seg

A relative path name of

dir5>old_seg

would designate the segment

>dir1>dir2>dir3>dir4>dir5>old_seg

A relative path name of

<dir0>newer

would designate the segment

>dir1>dir2>dir3>dir0>newer

A relative path name of

<<<sample_dir>game_dir>chess

would designate the segment

>dir1>sample_dir>game_dir>chess

The Star Convention

The asterisk character (loosely called a star) is used to designate groups of entries (in a single directory) which have similar names. This convention is applicable only in the final entry name of a path name. An asterisk in any component of an entry name matches (i.e., designates) any character string in that component position. Thus, a set of entries is specified. For example, the entry name

*.pl1

designates all two-component entries in the user's working directory which have pl1 as the second component;

sub_dir>my_prog.new.*

designates all three-component entries in the directory sub_dir (which is immediately inferior to the working directory) which

have my_prog.new as the first and second components; and

    *

and

    .

designate, respectively, all one-component and two-component entries in the working directory.

A double asterisk, permitted only in the rightmost component of an entry name, matches any number of components (including zero) on the right of the entry name. For example,

    my_prog.**

designates all segments with my_prog as the first (and possibly only) component; and

    *.my_seg.**

designates all segments with two or more components of which the second is my_seg.

The entry name ** designates all entries in the specified directory.

The main use for the star convention is to perform commands on a set of entries with similar names; e.g., delete all segments with a first component of square_root or list all two-component PL/I source segments.

<u>The Equals Convention</u>

Some commands (e.g., rename) deal with pairs of entry names as arguments. An equal sign as a component of the second entry name of a pair means that the character string from the corresponding component of the first entry name is to be substituted for the equal sign. For example,

    rename random.data_base ordered.=

is equivalent to

    rename random.data_base ordered.data_base

and

    rename *.data_base =.data1

renames all two-component entry names with data_base as the second component to have, instead, the second component data1.

If an equal sign appears in a component for which there is no corresponding component in the first entry name, then that component (the equal sign) in the second name is discarded. That is,

    rename alpha beta.=.gamma

is equivalent to

    rename alpha beta.gamma

A double equal sign as the rightmost component of the second entry name of a pair is equivalent to the corresponding component in the first entry name and any components following it. For example,

    rename one.two.three 1.==

is equivalent to

    rename one.two.three 1.two.three

and

    rename sqrt.** square_root.==

renames all entries with a first component of sqrt to have the first component square_root.

Any components appearing after the double equal are ignored. For example,

    rename aa.bb.cc dd.==.ff

would result in the entry dd.bb.cc since the ff is dropped.

Reference Names

Procedures executing in a process need to refer by name to other segments known in that process. Such a name is a reference name. A reference name may be the same as an entry name of the segment, or may be different. For example, when a dynamic linkage fault occurs for a reference name, the linker searches (using search rules) for a segment which has an entry name identical to that reference name. A procedure call, an invocation of a command through the command processor, or a reference to an external data segment is of this type, as is a segment made known by the hcs_$make_ptr subroutine. Search rules (telling which directories to search for the entry name) may be specified by the user or may be system defaults. The default search rules are described in the MPM Reference Guide section, The System Libraries and Search Rules. Alternatively, the user may explicitly designate the reference name to be associated with a specified segment. The initiate command and the hcs_$initiate and hcs_$initiate_count subroutines perform this function. In this case, the reference name need not have any similarity to any entry name of the segment.

Since a reference name is associated only with segments made known in a process, the same reference name may be used in two different processes to refer to two different segments. Also, a reference name/segment binding exists only for the duration of the process in which it is specified. It is possible to break that binding by terminating the segment, thus causing all links to that segment to be unsnapped and causing the segment to no longer be known in the process (by any reference name). The reference names of a terminated segment may be used again in the process to refer to a different segment. (See the write-up for the terminate command and the term_ subroutine.)

Individual reference names may be unbound in a process without terminating the segment unless the reference name removed was the only one on the segment. Note that no links are unsnapped so that previous connections made to a segment using that reference name remain in force.

Offset Names

Procedures frequently have more than one entry point, and data segments frequently have internal locations which are known externally by symbolic name. The names of the entry points and the internal locations are called offset names. Both designate symbolically an offset within the segment. The location specified may be referenced by the construction

ref_name$offset_name where the dollar sign separates the reference name and offset name.

In many cases the entry point to a procedure has the same name as the segment itself (or the segment has several entry names corresponding to the names of its entry points). A shorthand notation allows the offset name to be assumed to be the same as the reference name. For example,

```
call square_root (n);
```

is interpreted to mean

```
call square_root$square_root (n);
```

and the command line

```
rename a b
```

is equivalent to

```
rename$rename a b
```

It is worthwhile to remember that if the user has renamed one of his procedure segments (perhaps to preserve an old copy) or has linked to a segment using a different name, he must thereafter use the full reference name/offset name construction when referencing that segment as a procedure or external data segment. It is also important to note that if a reference name/segment binding has been established in a process, then merely renaming the segment will not break the association in that process. To do this, the segment must be terminated.

Command, Subroutine, Condition and I/O Stream Names

These names all have some conventions in common.

1) Each is permitted to be not more than 32 characters in length.

2) All ASCII characters are legal in any position except as noted in points 3 and 4 below.

3) System subroutine names will end in an underscore to prevent conflicts with subroutine names given by users. (I.e., the user may easily avoid conflicts by refraining from having an underscore as the last character of his subroutine names.)

4) Condition and I/O stream names which are part of the system should end in an underscore to help prevent conflicts with names given by users. A glance at the MPM Reference Guide sections, List of System Conditions and Default Handlers, and List of Names with Special Meanings, reveals many system condition and I/O stream names which do not observe this convention. These names were incorporated into the system before this convention was established, and changing them would be difficult.

(END)

SYSTEM PROGRAMMING STANDARDS

This section outlines many of the design and coding standards followed by Multics system programs. It is provided to give users some insights into what is considered to be good programming practice on Multics. The information presented below represents the accumulation of several years of experience in programming on Multics. It is hoped that it will aid users in their own programming efforts. As will be obvious, some of the standards apply only to modules of the system itself. On the other hand, those standards may suggest analogous procedures which would be appliable to other programming projects.

Coding Standards

1) All system subroutines must be pure, so that a single copy may be shared by all users. The Multics PL/I and FORTRAN compilers produce only pure subroutines.

2) All system subroutines must be written in the PL/I language. Explicit permission of the project management is required to use any other language. To aid others in understanding a program, the program listing should be well commented. This includes explaining the meaning of important variables.

3) Only subroutines documented as part of the Multics system (not including tools and the author-maintained library) may be called.

4) The names of all system programs that are not commands or active functions must end with an underscore (_). The names of all temporary segments and all I/O streams and condition names (other than PL/I defined condition names) used by system modules must also end in an underscore. This is to avoid naming conflicts with the user.

5) All variables used, including called subroutines, must be declared. This is done to increase program readability and reduce the confusion introduced by default or implicit declarations. For called subroutines, the parameter list must be fully declared, unless, of course, the subroutine accepts a variable number of arguments (e.g., a free format output subroutine). For readability, declarations should be collected together in a logical way (e.g., at the beginning of the subroutine or block for which they apply, or at the end) rather than being scattered throughout the program.

6) The use of pointers as arguments should be avoided when practical. Passing a data item as an argument rather than a

pointer to that item makes a program less error prone since the compiler can make checks for argument mismatch and since it is sometimes possible to perform run-time argument validation.

7) Special characters should be placed in the program directly. To lessen dependencies on the character code being used, the built-in function unspec should not be used for this purpose. For example,

```
declare nl (char(1) initial ("
");
```

declares "nl" to be a one-character string whose value is the new line character. The statement

```
unspec(nl) ="000001010"b;
```

should not be used.

8) Use of implicit conversion from one data type to another is prohibited, since it makes a program harder to understand. For example,

```
declare x fixed bin(18), y bit(18);

y=x;
```

should not be used. Instead one should write

```
y=bit(x,18);
```

9) Use of external static variables which do not contain a dollar sign (e.g., declare x external static) is prohibited since this data type is not efficiently implemented in the current Multics environment. External references of the form a$b are allowed. If the programmer needs to have an external data base which is shared among many subroutines, he may either create a segment by an appropriate storage system call and reference it using based structures or use the assembler to create a data segment by appropriate use of the segdef pseudooperation. The programmer wishing to do this should consult with a knowledgeable member of the Multics Development Group.

10) All variables should be of the automatic storage class unless there is a good reason for them to be internal static; i.e., they are static by nature. See also rule 11 below.

11) In PL/I programs, to avoid having to initialize variables whose values are constant every time the subroutine containing them is entered, and to avoid having copies of these variables made for every user of the subroutine, one should use internal static and initialize the variables using the initial attribute. The PL/I compiler will allocate space for these variables in the text section of the subroutine being compiled and will initialize them. Since the text section is pure, one copy of these variables will be used by all users of the subroutine. Unfortunately, if a variable of this type is passed as a argument to another subroutine, the compiler has no way of knowing whether or not that variable is to be changed by that subroutine and it, therefore, puts the variable into the linkage section. Therefore, if one has a large number of "constant" variables that are also passed as parameters, one should put them in the text portion of an assembly language program and initialize them using the appropriate data generating pseudooperations and reference them using either based structures or the "a$b" notation. This will assure that only one copy of these variables is used by all users of the subroutine. The programmer wishing more clarification of this point should consult with a knowledgeable member of the Multics Development Group.

12) Use of the PL/I <u>allocate</u> and <u>free</u> statements should be cleared in advance with project management, since there often exist more efficient ways to accomplish the same task. Subroutines that do perform allocations (or call subroutines which do) must establish a cleanup procedure to free the storage in the event that processing is aborted.

13) When possible, the PL/I on, revert and signal statements should be used instead of the condition_, reversion_ and signal_ subroutines since they are more efficient and make the program less system dependent.

14) The programmer should avoid writing PL/I functions with multiple entry points which return different data types unless there is a good reason to do so, since this generates extra code at each return statement.

Programming Style

1) The most common route through a program should be the most efficient. More exotic facilities which are inherently expensive should be separated from the simple facilities so that a casual user need not pay for the exotic each time he uses the simple.

2) System programs should, in general, use one of the three standard I/O streams: user_input, user_output, and error_output. Only special I/O service programs should issue I/O attach or detach calls for these streams. Commands should not, in general, provide optional off-line output. The file_output command is provided for this purpose.

3) All programs that are not commands or active functions should return a status code indicating successful completion or occurrence of an unexpected event, unless they are programs for which errors are unrecoverable or extremely rare; e.g., console output subroutines. This type of program should make use of the Multics signalling facility to signal that one-in-a-million error. In general, because of the higher overhead involved, programs should not make use of the Multics signalling facility for routine errors and status conditions. Subroutines which are directly called by the user must return only standard error_table_ codes. See the MPM Reference Guide section, Strategies for Handling Unusual Occurrences.

4) In most cases, programs that are not commands or active functions should not print error messages, but should allow a higher level subroutine to decide on the seriousness of errors and what to do about them. In general, it is wise to let the most qualified subroutine give the message. A good rule of thumb for determining the most qualified subroutine is to ask whether anything could be learned by reflecting the error to a higher level subroutine. If the answer is no, then the most qualified subroutine has been found.

5) All programs that are not commands, active functions or gates into a ring should assume they are called with the correct number and type of arguments and should not make checks. This is to avoid continually paying the cost of argument checking in programs which call the subroutines correctly. This does mean that the programmer must be careful to call subroutines correctly.

6) System programs should be prepared to execute properly even if they did not complete execution during a previous invocation

because of a quit or a fault. That is, they should either operate normally or warn the user of the consequences of continuing. For example, edm warns the user that, if he continues, the partically completed results of an earlier invocation will be lost.

7) System programs should never call a command if there is a subroutine which does almost the same thing. Commands are inherently more expensive since they are designed to interact directly with a human user.

8) System programs should not use a subroutine to do something which can be done reasonably easily in a few PL/I statements. The purpose of this rule is to avoid the proliferation of unnecessary system subroutines. The exceptions to this rule are input/output (see paragraph 1 under Error Handling and I/O below) and conversion from character to numeric data types. The reason for the latter exception is that this type of conversion is inherently more expensive than calling a specialized subroutine.

9) Calls to subroutines which require descriptors should be minimized when this does not conflict with program readability or degrade the user interface. This is because of the higher overhead involved in setting up argument lists with descriptors. For example, one should try to minimize the number of ioa_ calls in a program. This should not be interpreted to mean that one should remove all error messages from his program or make their output so terse as to be unreadable. It simply means that if, subject to the constraints mentioned above, it is possible to use one ioa_ call rather than two then the programmer should to so.

Data Base Management

Designing a program for a virtual memory environment requires a new outlook on program and data organization. Though the programmer is freed from the onerous task of allocating physical storage for his programs and data (e.g., storing intermediate results on secondary storage, overlaying parts of his programs with others to fit into core memory, etc.) he cannot ignore the issues of data management and program organization if he wants his program to be reasonably efficient. This is especially true for programs which manipulate large amounts of data. The attitude that an infinite virtual memory is available and if a program needs more room it can create another segment, may be all right for the casual user building a one-shot program

but not for the systems programmer. A major aim of the programmer should be to minimize the working set of his programs; i.e., his programs should create as few segments as is practical, reuse the ones they do create and should avoid unnecessary moving of data. The term working set is used loosely here to denote both the number of segments and the number of pages in the execution path of a program. In Multics it generally pays to spend CPU time (within reason) to save space. This principle should not, of course, be taken to an extreme. It does not mean, for instance, that one should not use a hash table. It is true that a hash table takes up more space than an equivalent linear list but a program will take fewer page faults referencing the former than searching the latter. In this case, the actual working set of the former is smaller even though its potential working set is larger. In all cases, the programmer must exercise his judgement as to the proper tradeoff between working set size and CPU usage, always avoiding the temptation to allow his working set to expand to infinity.

In addition to this basic principle, the following guidelines apply:

1) System programs must leave their data bases in a consistent state; e.g., a program which changes the contents of a segment should reset the bit count of that segment when it is finished. Programs should make any period of inconsistency as short as possible. They must also clean up after themselves; e.g., free storage should be released.

2) In order to assure consistent behavior, all standard translators must use the subroutine tssi_ to interface with the storage system. It might not make sense for nonstandard translators such as BASIC to use tssi_. Exceptions of this sort should be cleared in advance with the project management.

3) System programs should initiate the segments they access by a null reference name and should subsequently access those segments via a pointer. In general, segments initiated by a module should be terminated by that module (see point 4 below).

4) In general, the process directory should be used to hold temporary segments. If a program is not being entered recursively it should create temporary segments with intelligible names (e.g., containing the name of the creating program). It should clean up after itself before exiting by either truncating or deleting these temporaries. If the temporary segment can be reused the next time the program is

invoked it should be truncated; otherwise, it should be deleted. If a program is being entered recursively (e.g., one quits out of a command, issues a hold command, and reenters that command), it should create temporary segments whose names consist of a unique first component followed by one or more intelligible components. These segments should be deleted when the program exits. If, for some reason, a program cannot be made recursive it should detect the fact that it is being entered recursively, warn the user that partially completed work of an earlier invocation will be lost if he continues, then give him the option of continuing or exiting. Programs which create temporary segments should establish cleanup procedures to truncate or delete these segments if execution is abnormally terminated. As mentioned above, the names of temporary segments must end in an underscore.

5) Any system program which creates new segments (other than temporary segments) should put them into the user's current working directory unless the program explicitly makes provision for the user to provide a target directory. (The move and copy commands fall into this latter category.) The aim of this rule is to avoid messing up another directory, such as the directory from which a source segment was obtained.

6) System programs which create new segments must set access control lists according to the conventions enumerated below. If a segment is being replaced instead of being newly created, the command must leave the access control list as it was before the command acted. For instance, a translator finds that an object segment already exists with read and execute access for this user, and with other access for other users. The translator must obviously add write access to change the segment contents, but should restore the entire access control list to its former value when the translation is completed. The storage system interface subroutine tssi_ does this automatically for the translator writer. The access to be given to the user creating a segment is:

Segment Type	Access	Ring Brackets
directory segment	SAM	v,v
object segment	RE	v,v,v
data segment	RW	v,v,v

where v is the current validation level of the user. See the MPM Subsystem Writers' Guide section, Intraprocess Access

Control (Rings), for a discussion of validation level.

Additional Standards for Commands and Subsystems

Through the mechanism of the command processor any program -- system subroutine, system command, user subroutine -- can be invoked from the console. System commands are a special class of subroutines that are explicitly programmed with the console user in mind. They must check carefully for argument validity; they must warn the user of possible misunderstandings; they must be very reliable. They must, to the greatest possible extent, be a self-consistent set; i.e., the behavior of a command should be predictable from that of other commands.

For these reasons a number of additional standards are necessary for system commands and subsystems.

Naming Conventions

1) For ease of typing, all commands must have an abbreviated name consisting of the first letter of the first two or three syllables or first two or three words of its name (e.g., rename rn, unlink ul, print_attach_table pat).

2) All command names and abbreviations must be cleared in advance with the project management.

Programming Style and User Interface

1) If a command would also be useful as a subroutine, break it apart into a command which interfaces with the user (processes multiple arguments, handles the star and equals conventions, interprets control arguments, etc.) and a subroutine which does the work. This subroutine, like all subroutines, should return a status code rather than printing an error message. The outputing of error messages like all other user interface problems should be handled by the command.

2) Any command for which the star convention makes sense should use the star convention. Any command for which the equals convention makes sense should use the equals convention. See the MPM Reference Guide section, Contructing and Interpreting Names for a discussion of the star and equals conventions.

3) Characters which have special meanings to commands (e.g., "*", "=", ">" "<") should not be used in any context other than their standard one. For example, a command should not

interpret an argument of "*" as meaning that user wants to be logged out.

4) Commands should not be too powerful, that is, typing errors should not cause disastrous results. For example, with the old remove command

```
remove a>b
```

would delete the segment b in directory a, whereas

```
remove a> b
```

(i.e., one accidentally types a space before the b) would delete the directory a. To remedy this, there are now two commands: delete which deletes only nondirectory branches, and deletedir which deletes only directory branches.

5) Unless the purpose of a command is to produce some sort of output, it should produce no output during normal operation; i.e., it does not need to tell the user that it is doing its job. For example, if one enters the command

```
delete x y
```

the delete command produces output only if it has trouble deleting x or y. It does not type "deleting segment x", "deleting segment y". Commands which take a long time to execute (e.g., pl1) should print a short message when they are entered to indicated they are functioning. The general idea here is to reassure the user that he has not done something wrong. After more than a couple of seconds wait, the user, particularly a novice user, begins to worry that perhaps the computer is waiting for him.

6) Commands which take segment names as arguments should accept pathnames, not reference names, unless they explicitly deal with reference names (e.g., terminate_refname). The user who has a reference name he wishes to pass to a command may use the get_pathname active function to convert this reference name to a pathname (e.g.,

```
status [get_pathname x]
```

will cause the status command to be called with the pathname of the segment whose reference name is x). See the MPM Reference Guide section, Constructing and Interpreting Names

for a discussion of reference names.

7) Commands which interact with the typist should be prepared to handle the program_interrupt condition which is signalled by the program_interrupt command. Handling this condition correctly is quite tricky. See the MPM Reference Guide section, List of System Conditions and Default Handlers for details.

8) When a command which interacts with the typist produces an error message which the typist may not have expected, the command should normally follow the error message with a call to ios_$resetread (which discards all input read but not yet used) on the I/O stream from which it reads input so that the typist can modify his subsequent input.

9) We come now to a standard that is difficult to express with any degree of exactness. The phrase "commands should be designed with the user in mind" expresses the spirit of the standard. What follows is a series of examples designed to sensitize the reader to some of the issues involved in designing a command. Calling sequences should be logical (e.g., the user should not have to remember that % as a third argument to the xyz command causes all segments with a second component name fred to be deleted, whereas a ? in the same position suppresses this feature). Commands should allow the user to decide whether a protected segment should be deleted, rather than forcing him to make the segment deletable and to resubmit the delete request (or worse, delete the segment without warning). Judicious use of red console output is encouraged. It should be used to call attention to important or unusual occurrences. Remember, over-use destroys the whole purpose of red output -- a command which outputs everything in red may as well output everything in black. Canned messages printed by commands should not contain characters which come out as escape characters on IBM model 1050 and model 2741 consoles and on model 37 teletypes (e.g., "¢<segment¢> not found" is not an acceptable message).

<u>Argument Handling</u>

1) Commands, wherever possible, must accept path names (not just entry names) as arguments. The subroutine expand_path_ should be called to convert a relative path name into an absolute path name.

2) Commands which deal with segments whose names have a fixed suffix should not force the user to type that suffix.

Rather, they should append that suffix to their arguments if it is not given. For example, the command lines

pl1 x

and

pl1 x.pl1

should be equivalent.

3) Commands whose interface is simple (such as the delete and addname commands) should accept multiple arguments if it makes sense to do so.

4) All commands which accept a variable number of arguments should declare themselves as having no arguments (i.e., command_name: proc;) and should obtain their arguments using the procedure cu_$arg_ptr.

5) Commands must obey Multics control argument conventions as described in the MPM Reference Guide section, List of Command Control Arguments.

6) In general, for the convenience of the user, command arguments should be order independent unless the order dependency serves a useful purpose (as in the -ag control argument of the enter_abs_request command).

<u>Error Handling and I/O</u>

1) The input/output facilities of the PL/I language must not be used in system programs since they are more expensive than system-provided subroutines.

2) To read a line from the input stream user_input, use the subroutine ios_$read_ptr. To read a line with appropriate data type conversion (i.e., the user is typing in pointers, floating point numbers, etc.) use the subroutine read_list_.

3) Output lines fall into three distinct classes:

a) unusual status messages
b) questions
c) everything else

Lines of type a) should be output using the subroutines com_err_ and active_fnc_err_ (active functions should use active_fnc_err_, all other modules should use com_err_). Lines of type b) should use the subroutine command_query_. These three subroutines are provided in order to centralize the processing of lines of type a) and b) so that changes in system conventions in this area may easily be made. For lines of type c) the subroutine ios_ should be used when it is necessary to format an output line; otherwise, use the subroutine ios_$write_ptr.

4) Commands should check for status codes which have special meaning to them and either print appropriate error messages or, if the error is easily recoverable, allow for user intervention using command_query_. All such messages must contain the name of the command which generated them, since otherwise the user would have no way of knowing which command generated a given message if he has issued several at once or was running an exec_com segment. Complex programs such as compilers may output diagnostics by standard output subroutines but should have at least one call to com_err_ to notify the system that an error has occurred.

(END)*

ACCESS CONTROL

Access control is the regulation of the right of a process (the active component of the system) to use or reference objects within the system. Examples of such objects are typewriters, printers, segments, and processes. This section discusses the regulation of the right of processes to use or reference certain objects within the Multics storage system, namely directories and segments.

This section is divided into two parts. The first part explains what rights may be granted or denied a process referencing a segment or directory. The second part describes how different access rights may be granted to different processes, i.e., interprocess access control.

A few sentences are in order about the use of this section. The access control mechanism represents an attempt to provide a general capabilty for controlling access in many different ways and yet keep the mechanism simple for common applications. This section is a comprehensive description of the full access control mechanism and most readers will find much if not all of the material of no interest to them. Users who do no sharing of segments, i.e. those who have segments which only they reference, need not know anything about access control because the system defaults automatically provide for this case. Even if the user makes use of programs of other users he need not know anything about access control because setting access is the responsibility of the other users. Only if the user wishes to share his segments with other users need he know anything about access control. In this case he should first read the MPM Introduction Chapter 3, Beginner's Guide to the Use of Multics. That chapter provides sufficient information about access control for most common applications. Only if that chapter is insufficient for the user's needs, should he then read this section.

Yet another facet of Access Control is described in the MPM Subsystem Writers' Supplement section Intraprocess Access Control (Rings). This part of access control differentiates between the access rights that a process may be granted in different states. It is called the ring mechanism, and is of use to the subsystem writer who wishes to write a protected subsystem.

Part 1: Access Modes

One does not simply want to regulate whether or not a process can reference a given object, but usually wants a finer control in order to regulate various ways in which a process may use an object. For different types of objects the means of

referencing may be different. For segments and directories these ways of referencing objects are termed modes of access or access modes. Since segments and directories are different types of objects, having different properties and different operations for referencing them, they have different modes.

Segment access modes determine the ways in which a process may reference the data of a segment. Directory access modes determine the ways in which a process may reference the attributes of directory entries. Each mode is labelled by a distinct, single character identifier that is used when specifying the mode to system commands.

The access modes for segments are:

execute (e) an executing procedure may transfer to this segment and words of this segment may then be interpreted as instructions and executed by a processor;

read (r) the process may execute instructions that cause data to be fetched (loaded) from the segment;

write (w) the process may execute instructions that cause data in the segment to be modified. *

The access modes for directories are:

status (s) the attributes of segments, directories and links contained in the directory and certain attributes of the directory itself may be obtained by the process (see the MPM Reference Guide section on Segment, Directory and Link Attributes for a definition of attributes);

modify (m) the attributes of existing segments, directories and links contained in the directory and certain attributes of the directory itself may be modified; and existing segments, directories, and links contained in the directory may be deleted;

append (a) new segments, directories and links may be created in the directory.

If a segment or directory is not accessible in any of the above modes then the process has no access to the segment.

* Until step 3 of directory reformatting has been completed (probably about February, 1973), the segment access mode append (a) should appear on segment ACLs that have write (w) access mode.

Part 2: Interprocess Access Control

In order to be able to grant different processes distinct access rights it is necessary to be able to distinguish different processes. For this purpose, each process has an associated access identifier. The access identifier is fixed for the life of the process. The identifier is a three component character string with the components separated by periods (.). The first component is the name of the person on whose behalf the process was created. The second component is the name of the project group of which the person named in the first component is a member. This person-project combination is termed a user. The same person may log into Multics under different projects and is considered to be two different users. The third component of the access identifier is the instance which is a single character used to distinguish different processes belonging to the same user. The access identifier must be less than 33 characters in length. The access identifier Jones.Faculty.a would be associated with a process created for Jones in the Faculty project. The "a" instance distinguishes the process from another process created for Jones.Faculty which might have an access identifier Jones.Faculty.b. All processes need not have distinct access identifiers. It is quite likely that several processes have the access identifier Jones.Faculty.a which simply means that all these processes have the same access rights to segments and directories in the storage system.

Access Control List

The rights that different process have when referencing a segment or directory are specified as an attribute of that segment or directory in the form of a list called the Access Control List (ACL). Each entry of the list specifies a set of processes (actually a set of access identifiers of processes) and the access modes that members of that set may use when referencing the segment or directory. The modes read, write, and execute may be specified in ACLs of segments and the modes status, modify, and append may be specified in ACLs of directories. On directory ACLs, modify mode may not appear without status mode. If some of these access modes are not granted in a ACL entry, then processes specified in the entry cannot access the segment or directory in the ungranted mode. For example, if the ACL of a segment contains an entry for a process and the modes specified are read and execute then the given process may execute instructions that fetch data from the segment, and transfer to and execute instructions in the segment, but it may not modify data in the segment.

The members of the set of processes associated with an ACL entry are specified by a character string called a process class identifier. The process class identifier is similar in appearance to an access identifier. In fact a string which is an access identifier may also be a process class identifier. Such a process class identifier identifies the class of processes whose access identifiers are the same as the process class identifier; e.g., the process class identifier Jones.Faculty.a identifies the class containing all processes with access identifier Jones.Faculty.a.

It is very useful to identify larger groups of processes than simply those with the same access identifier. This may be accomplished by replacing one or more of the three components of the process class identifier (i.e., the person name, project name, or instance) by the asterisk character (*). Such a character string identifies that class of processes whose access identifiers match the remaining components of the character string; i.e., those components of the string that are not the asterisk character. For example, the class identifier Jones.*.a identifies that class of processes with an access identifier containing Jones as the person identifier and "a" as the instance. Any project identifier in the access identifier will match. Therefore, processes with access identifiers Jones.Work.a, Jones.Lazy.a, and Jones.Faculty.a will be members of the class identified by Jones.*.a. Similarly, processes with access identifiers Jones.Lazy.a, Jones.Work.q, and Jones.Faculty.q are members of the class identified by Jones.*.*. The string *.*.* identifies the class of all processes.

<u>Structure of an Access Control List</u>

From the above discussion one can see that it is quite possible for a single process to be a member of more than one process class. This situation can lead to ambiguities on ACLs when more than one entry can apply to the same process. To eliminate this ambiguity and make ACLs more easily readable, four conventions are imposed on ACLs and their interpretation. First, no process class identifier may appear more than once on any ACL. Second, the ACL is ordered as explained below. Third, the entry that applies to a given process is the first entry on the list whose process class contains the given process. Finally, if no entry exists on the list for a given process then that process has no access to the segment or directory. These conventions assure that the access for every process is uniquely specified by the ACL.

In order to properly generate and modify ACLs it is necessary to have some understanding of how they are ordered. The ordering is done by leftmost specificity of components of process class identifiers. This can be easily explained by a simple ordering algorithm and an example. The entries to be ordered are first divided into two groups, those whose first (person) component are specific (i.e., are not asterisk) and those whose first component are asterisk. Those with specific first component are placed first on the ACL. Within these two groups a similar ordering is done by second (project) component again with the specific entries being first. This produces four groups. Finally, within each of these four groups a similar ordering is done on the third (instance) component to produce eight groups. The eight groups resulting will be in the following order:

1) class identifiers with no asterisks

2) class identifiers with an asterisk in the third component only

3) class identifiers with an asterisk in the second component only

4) class identifiers with asterisks in the second and third components only

5) class identifiers with an asterisk in the first component only

6) class identifiers with asterisks in the first and third components only

7) class identifiers with asterisks in the first and second components only

8) the class identifier *.*.*

Within each of these groups the ordering is unimportant because a process may belong to only one class in a group. The following is a validly ordered ACL:

Jones.Work.a	r	(1)
Smith.Lazy.*	rw	(2)
White.*.q	re	(3)
Black.*.*	rew	(4)
*.Faculty.m	no access	(5)
.Student.	re	(6)
.Lazy.	r	(7)
..b	rew	(8)
..*	r	(9)

In the above example a process with access identifier Smith.Lazy.h would be able to read and write the segment as derived from entry (2), a process with access identifier Jones.Lazy.h would be able only to read the segment as derived from entry (7), and a process with access identifier Smith.Faculty.q would be able to read the segment as derived from entry (9). Note that despite entry (9), which apparently grants read access to all processes, Smith.Faculty.m has no access since entry (5) is encountered first.

Maintenance of Access Control Lists

Both commands and subroutines are provided for the purpose of creating and modifying ACLs. The commands are listacl, setacl, and deleteacl (see the MPM write-ups for these commands). The subroutines are hcs_$add_acl_entries, hcs_$add_dir_acl_entries, hcs_$replace_acl, hcs_$replace_dir_acl, hcs_$delete_acl_entries, hcs_$delete_dir_acl_entries, hcs_$list_acl, and hcs_$list_dir_acl (see the MPM write-ups for these subroutines). The specific usage of each of these procedures is described in their command and subroutine write-ups. The commands and subroutines enforce the constraints mentioned above; i.e., they order the ACL and do not permit more than one entry with a given process class identifier to appear on the ACL.

Consider the example of a segment with an ACL containing the single entry:

Jones.*.*	r

A new entry is added for the process class *.Work.* resulting in the ACL:

Jones.*.*	r
.Work.	rw

This would superficially appear to given all members of the Work project the right to read and write the segment. In actuality it gives all members of the Work project the right to read and write the segment except for Jones (assuming Jones is a member of the Work project). Jones has only read access. If we truly wanted to give <u>all</u> members of the work project write access we would have to add another entry to produce:

Jones.Work.*	rw
Jones.*.*	r
.Work.	rw

The entry Jones.*.* is still useful for specifying access for Jones when he logs in on any project other than Work.

It is important to realize that placing a new entry on an ACL does not necessarily grant all members of that process class the specified access, for some members of that process class may also be members of process classes appearing earlier on the ACL. The user should, therefore, be aware of what an ACL currently contains before modifying it.

Special Entries on Access Control Lists

Several Multics system services are performed by special processes as opposed to being done in the user's process. These system service processes perform such functions as making backup copies of segments in the storage system and queued printing and punching of segments at users' requests. In order for these service processes to perform these functions they must have access to the segments to be serviced. In many cases the service processes normally service all segments in the storage system and, therefore, need access to most segments. These service

processes and only these service processes are members of a single project called SysDaemon. In order to assure that these service processes have access to the segments the storage system subroutines automatically place the ACL entry

    .SysDaemon.   rw

on the ACL of every segment, and the ACL entry

    .SysDaemon.   sma

on the ACL of every directory when the segment or directory is created or its ACL is entirely replaced. A user taking no special action with regard to any members of the SysDaemon project will, therefore, have automatically granted the necessary access to all service processes so that they may perform their function.

Under special circumstances, some user may elect not to receive the service of a service process on some of his segments. To do this, the user simply denies access to his segments to that service process by modifying the ACL to contain an entry for that service process with null access. It is crucial that a user who elects not to receive such a system service be fully aware of the nature of the service and the consequences of his choice. For example, if the backup processes are not permitted access to a segment, backup copies of the segment cannot be made and the segment will not survive certain types of system failure.

<u>Default Values for Access Control Lists</u>

Many system commands and subroutines, e.g., create, create_dir, and hcs_$append_branch, add an entry for the creating process to the ACL of a newly created segment or directory. The storage system subroutines also automatically add the above mentioned service process entry to all newly created segments and directories. It is also useful to be able to specify a list of entries to be added to all newly created segments in addition to entries for the creating process and the service processes. This eliminates the need to expicitly modify an ACL each time a new segment or directory is created. This list of entries to be added to newly created segments or directories is called an <u>initial access control list</u> or initial ACL and is an attribute of a directory. Each directory has two sets of initial ACLs, one set for segments appended to the directory and one set for directories appended to the directory. Since each initial ACL is simply a list of ACL entries, it has the appearance of an ACL. When a segment or directory is created the service process ACL

entry is first placed on the ACL of the segment or directory. Then the appropriate initial ACL (i.e., either the one for segments or the one for directories) of the containing directory is merged with the ACL. The merging of two ACLs means that the entries are combined and sorted. If two entries on the resulting ACL contain the same process class identifier, then the entry that was originally on the ACL of the segment is deleted leaving the newly added entry. In this way the service process entry originally on the segment may be overridden by the initial ACL by placing an entry with process class identifier *.SysDaemon.* on the initial ACL. Finally, any entries specified in the call to append the segment (for most system commands this is simply one entry for the creating process) are merged into the ACL. Again these entries will override the service process and initial ACL entries if duplicate process class identifiers exist.

The default value for the initial ACLs of a newly created directory is empty, i.e., there are no entries in the initial ACLs.

<u>Reference</u>

Organick, E.I., <u>The Multics System: An Examination of its Structure</u>, Chapter 4, Access Control and Protection, M.I.T. Press, Cambridge, Mass. 1972

 (END)

INDEX

This Index covers only Part II of the manual, namely the Reference Guide sections 1 to 8, and the command and subroutine write-ups.

The Index is organized around the numerically ordered Reference Guide sections and the alphabetically ordered commands and subroutine write-ups, rather than by page number. Thus, for example, the entry for bulk input and output might read:

```
bulk I/O
   3.4
   4.4
   dprint
   dpunch
```

The first two items under bulk I/O refer to the Reference Guide sections 3.4 and 4.4, and the last two to the write-ups for the dprint and dpunch commands. They are referenced in the order that they appear in this manual. Note that command names can normally be distinguished from subroutines by the trailing underscore in the segment name of subroutines.

Some entries are of the form:

```
I/O (bulk)
   see bulk I/O
```

For simplicity of usage, these entries always refer to other places in the Index, never to normal Reference Guide, command or subroutine write-ups.

Some entries are followed by information within parentheses. This information serves to explain the entry by giving a more complete name or the name of the command under which the actual entry can be found. For example:

```
e (enter)
listnames (list)
```

In addition to this Index, other indexes to information are:

1) MPM Table of Contents
   - lists names of commands and subroutines with write-up issue dates
   - lists commands and subroutines documented under other write-ups; e.g., console_output: see file_output

2) Reference Guide Section 1.1: The Multics Command Repertoire
   - lists commands by function

3) Reference Guide Section 2.1: The Multics Subroutine Repertoire
   - lists subroutines by function

4) Reference Guide Section 8.3: Obsolete Procedures

2434 6